Git i GitHub

Kontrola wersji, zarządzanie projektami i zasady pracy zespołowej

—

Mariot Tsitoara

Helion

Apress

Tytuł oryginału: Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer

Tłumaczenie: Maksymilian Gutowski

ISBN: 978-83-283-8735-5

First published in English under the title Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer by Mariot Tsitoara, edition: 1

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/wprgit
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Commity

W poprzednim rozdziale omówiłem podstawowe funkcje Gita. Wiesz już, jak sprawdzić historię i przejrzeć zmiany wprowadzone w aktualnej wersji. Commity Gita są jednak twardym orzechem do zgryzienia, więc w tym rozdziale opowiem o nich więcej. Zaczniemy ponownie od tajników działania Gita i zaznajomienia się z terminologią, a następnie wyjaśnię, jak wywoływać i sprawdzać wcześniejsze wersje. Zaczynamy!

Trzy stany Gita

Zanim opowiem o szczegółach, wróćmy do podstaw i przypomnijmy sobie, jak Git działa. Z całą pewnością pamiętasz trzy stany, w jakich może się znajdować plik. Jeśli nie pamiętasz, nie pomijaj tego rozdziału. Ta wiedza jest nieodzowna w pracy z Gitem. Jeśli natomiast pamiętasz te stany, to także nie pomijaj tego rozdziału — poświęciłem naprawdę dużo czasu, by go napisać.

Jak pokazałem w poprzednim rozdziale, nie wszystkie pliki są śledzone w Gicie; niektóre z nich można ignorować za pomocą pliku *.gitignore*. Są też pliki, które nie są wprawdzie ignorowane, ale nie są też jeszcze śledzone przez Gita. Mowa o nowo utworzonych plikach, które nie zostały ujęte w migawce (czy też commicie).

Śledzone pliki mogą się znajdować w jednym z trzech stanów:

- Zmodyfikowany — do pliku zostały wprowadzone zmiany.
- Dodany do przechowalni — do pliku zostały wprowadzone zmiany i został on przygotowany do uchwycenia w migawce.
- Zatwierdzony — została zachowana migawka całego projektu i plik został w niej uchwycony.

Nieśledzone pliki pozostają takimi, dopóki nie wyślesz ich do przechowalni i nie zatwierdzisz albo dopóki nie polecisz, aby były ignorowane.

Pamiętaj: Git nie rejestruje zmian, tylko migawki. Przy każdym commicie zapisany zostaje stan całego projektu, a nie wszystkie drobne zmiany, które wprowadzono.

Ciekawostka: Git działa szybko, ponieważ zawsze pracujesz na ostatnim zapisanym stanie projektu. Kiedy chcesz zobaczyć wcześniejszy commit, Git pokazuje stan projektu z określonej daty. Wiele systemów kontroli wersji zapisywało wszystkie wprowadzane zmiany, więc kiedy programista chciał przywrócić poprzedni stan, systemy te wprowadzały zmiany wstecz. Gdy projekt się rozrasta, taki sposób działania może być przyczyną wielu problemów z szybkością i pamięcią. Czy sposób, w jaki Git funkcjonuje, nie prowadzi do tworzenia olbrzymich baz danych? Nie, ponieważ kiedy wykonujesz migawkę, pliki, które nie uległy zmianie, nie zostają zapisane ponownie, lecz jest tworzony odnośnik do pliku.

Wróćmy jeszcze raz do tematu trzech stanów i spójrzmy na łączące je zależności:

- Pracujesz na katalogu roboczym. Jest to po prostu katalog, który utworzyłeś przed zainicjalizowaniem repozytorium. To właśnie w nim będziesz odczytywać i edytować pliki.
- Przechowalnia jest miejscem, do którego trafiają zmienione pliki przed utworzeniem migawki całego projektu. Nie możesz utworzyć migawki, jeśli pliki nie zostały wysłane do przechowalni. Jedynie pliki znajdujące się w przechowalni (i niezmienione pliki) zostaną uwzględnione w migawce. Pliki poza przechowalnią (śledzone lub nie) oraz ignorowane zachowają swój stan.
- Baza danych lub katalog *.git* przechowuje każdą wykonaną migawkę. Takie migawki nazywa się commitami.

Pamiętaj: przechowywanie dotyczy jedynie wybranych przez Ciebie zmienionych plików, a zatwierdzanie dotyczy całego projektu. Plik przechowujesz, a projekt commitujesz.

Poruszanie się po wersjach

Często będzie Ci zależało nie tylko na możliwości sprawdzenia zmian, jakie wprowadzono do projektu, ale także na przyjrzeniu się stanowi, w jakim się znajdował — zobaczeniu wykonanej migawki. W Gicie to wszystko jest proste.

Kiedy chcesz wywołać wcześniejszy stan projektu w katalogu roboczym, musisz przełączyć się na commit poleceniem `git checkout`. Ponieważ w wyniku tego pliki znajdujące się w katalogu roboczym ulegną zmianie, musisz się upewnić, czy nie znajdują się w nim przypadkiem pliki, które nie zostały wysłane do przechowalni. Pliki niesledzone mogą wciąż się w nim znajdować, ponieważ Git jeszcze nie rejestruje ich stanu.

Aby obejrzeć migawkę projektu, wydajemy polecenie `git checkout` wraz z nazwą commita jako parametrem:

```
$ git checkout <nazwa>
```

Wypróbujmy to! Otwórz swój bieżący projekt w edytorze tekstu i spójrz na jego zawartość. Sprawdź teraz poprzedni commit tak jak na rysunku 5.1.

■ **Ostrzeżenie** Nie możesz przełączyć się na inny commit, jeśli katalog roboczy nie jest pusty. Koniecznie zatwierdź wszystkie zmiany przed przełączeniem się na inną migawkę.

Uważaj, by przypadkiem niczego nie zmienić przy sprawdzaniu wcześniejszych commitów. To działa jak na filmach: lepiej nie zmieniać przeszłości!

Rysunek 5.1. *Sprawdzanie poprzednich commitów*

W edytorze tekstu zobaczysz, że projekt wygląda teraz dokładnie tak jak wtedy, kiedy wykonywałeś migawkę. Właśnie to jest najlepsze w Gicie. Nic, co zostało uchwycone w migawce, nie może zostać utracone!

Zapoznajmy się teraz z odrobiną terminologii Gita. Pierwsze pojęcie to „head", który jest po prostu wskaźnikiem do commita. Kiedy mówimy o commitach, nie nawiązujemy do nazwy, lecz do heada.

Gdy przełączamy się między różnymi commitami, musimy jakoś się orientować, na którym headzie jesteśmy. Aktualny head (ten, który przeglądamy) oznaczany jest po prostu jako *HEAD*.

I już! Head jest wskaźnikiem odsyłającym do commita (w repozytorium może się znajdować wiele headów), a head wskazujący na aktualnie przeglądany commit to *HEAD*.

Ćwiczenie: poruszanie się po historii

Przełącz się z jednego commita na inny poleceniem `git checkout`. Pod żadnym pozorem niczego nie zmieniaj.

Jak jednak wrócić do normalnego, bieżącego katalogu roboczego? Ponieważ nie wprowadziliśmy żadnej dużej zmiany do repozytorium, powrót do katalogu roboczego sprowadza się do przełączenia się na jedyną gałąź, jaką mamy. Zgodnie z konwencją nosi ona nazwę *master*:

```
$ git checkout master
```

Wypróbuj! Pamiętaj jednak o świętych zasadach podróży w czasie:

- Wolno cofać się w przeszłość tylko wówczas, jeśli w teraźniejszości panuje porządek (czyli nie ma żadnych niewysłanych do przechowalni plików w katalogu roboczym).
- Nie wolno zmieniać przeszłości (przynajmniej dopóki nie będziesz mieć większego doświadczenia).

Nie zapomnij o przełączeniu się na aktualną gałąź (*master*) po przeglądaniu starszych wersji.

Cofanie commita

Nadejdzie czas, kiedy wyślesz pliki do przechowalni i je zatwierdzisz, aby później się rozmyślić. Każdemu może się to zdarzyć. W przypadku tradycyjnych metod (bez wersjonowania) cofanie zmian jest bardzo trudne, zwłaszcza kiedy ma się do czynienia ze zmianami wprowadzonymi dawno temu. W Gicie wystarczy jednak użyć jednego polecenia: `git revert`.

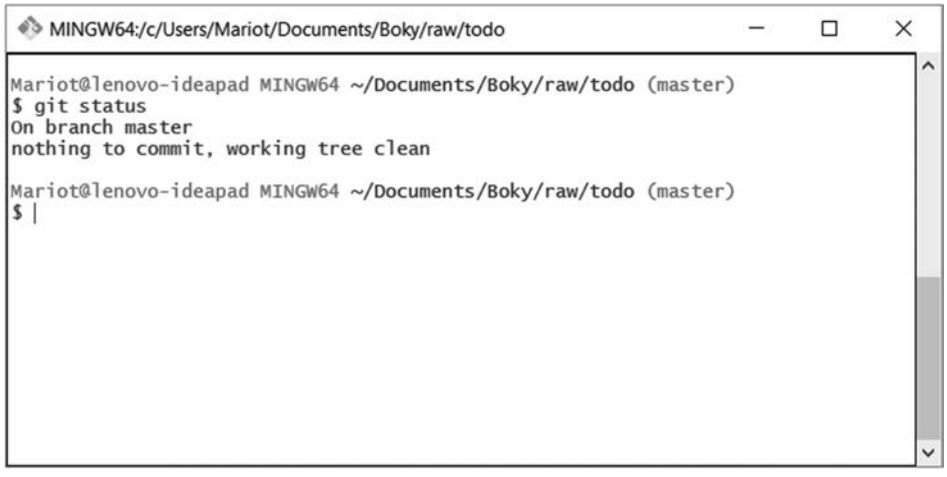
Czy nie lepiej byłoby usunąć commit? Nie, ponieważ obowiązuje zasada podróży w czasie z poprzedniego punktu: nie wolno zmieniać przeszłości. Zatwierdzone zmiany muszą takimi pozostać dla dobra historii; zmiana tego, co stało się w przeszłości, jest bardzo niebezpieczna i w dodatku niezgodna z intuicją. Zamiast tego należy użyć polecenia `git revert` do utworzenia nowego commita, będącego przeciwieństwem commita, który chcesz cofnąć.

Cofnięcie commita jest zatem zatwierdzeniem jego przeciwieństwa. To takie proste! Aby użyć tego polecenia, musisz podać nazwę commita jako parametr:

```
$ git revert <nazwa commita>
```

Możesz cofnąć dowolny commit. Musisz jedynie się upewnić, że pracujesz na czystym katalogu roboczym. Nie zapomnij zatem o wysłaniu plików do przechowalni i zatwierdzeniu ich przed cofnięciem commita. Wypróbujmy to!

Na początek musisz sprawdzić, czy katalog roboczy jest czysty (rysunek 5.2).

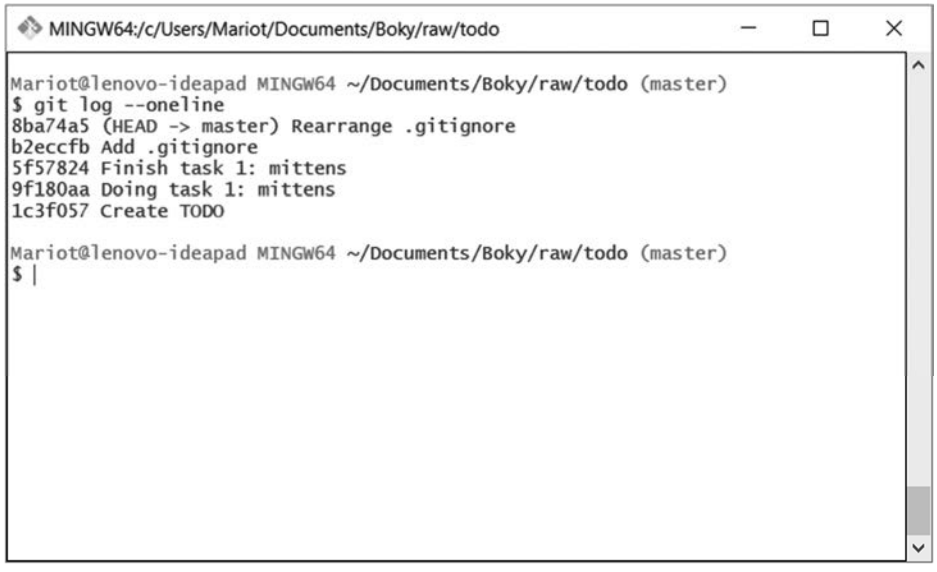


Rysunek 5.2. *Sprawdzenie katalogu roboczego poleceniem git status*

Doskonale. Skoro już wiemy, że katalog roboczy jest czysty, nadszedł czas, aby przejrzeć historię i zorientować się, który commit należy cofnąć. Powinniśmy otrzymać rezultat taki jak na rysunku 5.3.

Rysunek 5.3. *Sprawdzenie historii commitów poleceniem git log*

■ **Uwaga** Jeśli nie podoba Ci się format, w jakim przedstawiona jest historia commitów, możesz podać parametr `--oneline`, aby ograniczyć ilość wyświetlanych informacji. Przykład widać na rysunku 5.4.



Rysunek 5.4. *Bardziej estetyczny wynik polecenia git log*

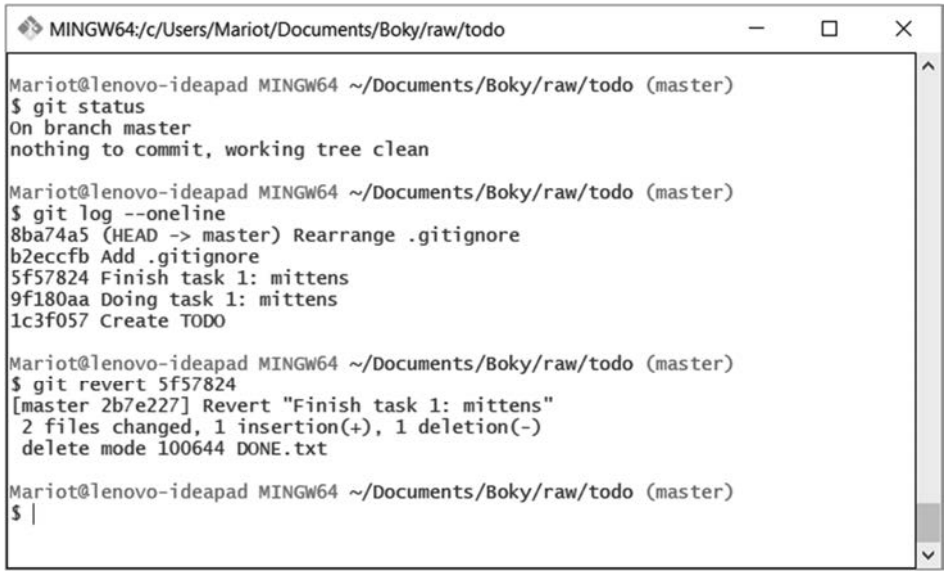
Przywróćmy trzeci commit! Wystarczy użyć polecenia `git revert` z nazwą commita:

```
$ git revert 5f57824
```

Ponieważ `git revert` tworzy jedynie nowy commit ze zmianami przeciwnymi do wcześniej wprowadzonych, cała procedura wygląda poza tym jak tworzenie nowego commita. Jak widać na rysunku 5.5, zostaniesz poproszony o opisanie nowego commita. Zalecam, aby zawsze zachowywać domyślny opis commita, gdyż ułatwia to jego identyfikację.

Rysunek 5.5*. Opis nowego commita*

Po zapisaniu opisu commita (tak jak zwykle) otrzymujesz streszczenie zawartości migawki. Rysunek 5.6 przedstawia wynik, jaki uzyskasz po wykonaniu poleceń i zapisaniu opisu commita.



Rysunek 5.6. *Streszczenie cofnięcia zmian*

Jak widzisz, cofanie zmian jest w Gicie bardzo proste. Należy pamiętać, że `git revert` tworzy jedynie nowy commit ze zmianami przeciwnymi do tych, które wprowadzono wcześniej. Oznacza to, że możesz też cofnąć cofnięcie! Cofnięcie cofnięcia polega na ponownym zatwierdzeniu pierwotnego commita, a oba cofnięcia wzajemnie się neutralizują. Same commity jednak pozostaną w historii, ponieważ nie możesz zmienić przeszłości.

■ **Uwaga** Tak naprawdę możesz zmienić przeszłość. Nigdy jednak tego nie rób. To bardzo zły pomysł, którego skutkiem jest masa problemów.

Modyfikowanie commita

Jak obiecałem Ci w poprzednim rozdziale, dowiesz się teraz, jak zmodyfikować commit. Możesz to robić, jeśli zapomniałeś o wysłaniu pliku do przechowalni lub jeśli chcesz zmienić opis commita. Nie powinno się modyfikować wielu plików, jako że działanie tej funkcji jest mało intuicyjne. W kolejnym rozdziale szczegółowo omówię, kiedy i gdzie należy to robić. Powtarzam po raz kolejny: pod żadnym pozorem nie próbuj zmieniać przeszłości.

Aby zmodyfikować commit, musisz użyć polecenia `git commit` z parametrem `--amend`. Otworzysz w ten sposób domyślny edytor tekstu tak jak w przypadku zwykłego commita, z tym że wysłane do przechowalni pliki i opis commita już tam będą.

```
$ git commit --amend
```

Wystarczy następnie zapisać plik i zamknąć edytor tekstu, tak jak zwykle. Pojęcie „modyfikacja" jest nieco mylące, ponieważ nie modyfikujesz commita, tylko tworzysz nowy i zastępujesz aktualny. Od tej pory będę zatem używał słowa „poprawianie".

Poprawienie commita polega na wzięciu wszystkiego z przechowalni i utworzeniu nowego commita. Jeśli zatem chcesz dodać nowy plik do commita lub jakiś z niego usunąć, możesz w tym celu do woli poprzenosić pliki z przechowalni i do niej. Pamiętaj: aby wycofać plik z przechowalni, musisz użyć polecenia `git reset HEAD <nazwa_pliku>`. Poniżej zobaczysz przykład.

Sięgnijmy ponownie po swoją listę zadań do wykonania. Edytuj istniejący plik, a następnie utwórz dwa nowe pliki o nazwach *filenottocommit.txt* i *fileforgotten.txt* tak jak na rysunku 5.7.

Name	Date modified	Type	Size
another	2019-05-27 22:20	File folder	
build	2019-05-23 21:57	File folder	
folder	2019-05-23 22:08	File folder	
.gitignore	2019-05-27 22:20	Text Document	1 KB
DOING.txt	2019-06-17 23:11	Text Document	1 KB
DONE.txt	2019-08-12 23:45	Text Document	0 KB
fileforgotten.txt	2019-08-12 23:46	Text Document	0 KB
filenottocommit.txt	2019-08-12 23:46	Text Document	0 KB
PRIVATE.txt	2019-05-27 22:20	Text Document	0 KB
TODO.txt	2019-06-17 23:31	Text Document	1 KB

Rysunek 5.7. *Wszystkie pliki w katalogu roboczym*

Możesz sprawdzić aktualny stan projektu poleceniem `git status`:

```
$ git status
```

W zależności od tego, ile plików dodałeś do tej pory do projektu, możesz uzyskać nieco inny rezultat, ale wciąż będzie podobny do tego z rysunku 5.8.

Kolejną czynnością, jaką należy wykonać, jest wysłanie plików do przechowalni, aby włączyć je do commita. Podaj nazwy zmienionych plików i *filenottocommit.txt*:

```
$ git add TODO.txt DONE.txt filenottocommit.txt
```

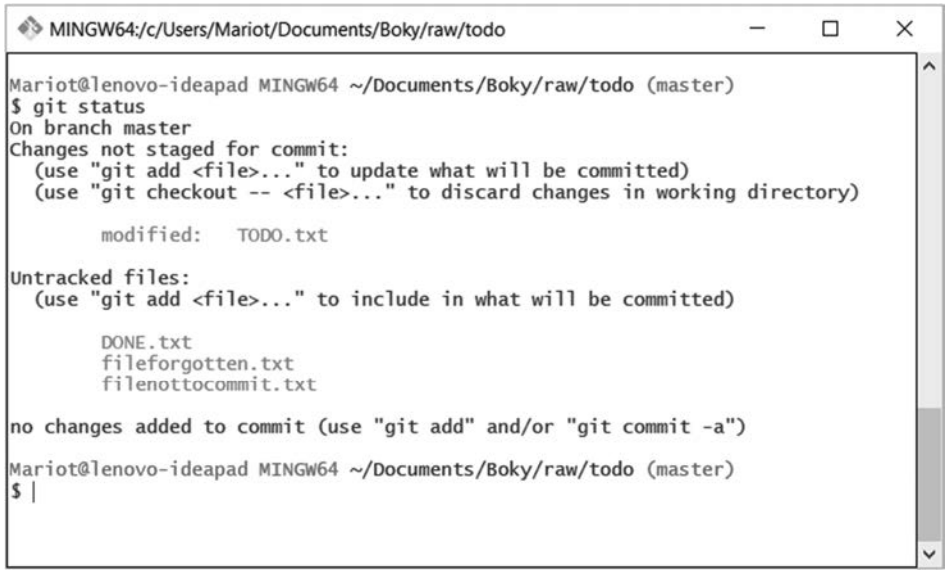
Z ostatniego rozdziału wiesz, że przed zatwierdzeniem zawsze trzeba sprawdzać, co wysłaliśmy do przechowalni, używając polecenia `git diff --staged`. Wyobraźmy sobie jednak, że zapomnieliśmy to zrobić i od razu zatwierdziliśmy commit.

```
$ git commit
```

W takim przypadku trafimy na komunikat opisujący zmiany do zatwierdzenia (rysunek 5.9).

Jak widzisz, wyróżnione są pliki do zatwierdzenia i nieśledzone. Dość trudno ich nie zauważyć, ale udajmy, że je przeoczyliśmy. Wpisz prosty opis commita, zapisz plik i zamknij edytor. Zobaczysz standardowe streszczenie widoczne na rysunku 5.10.

W streszczeniu commita widzisz, że zatwierdziłeś niewłaściwy plik, a zapomniałeś o zatwierdzeniu innego.

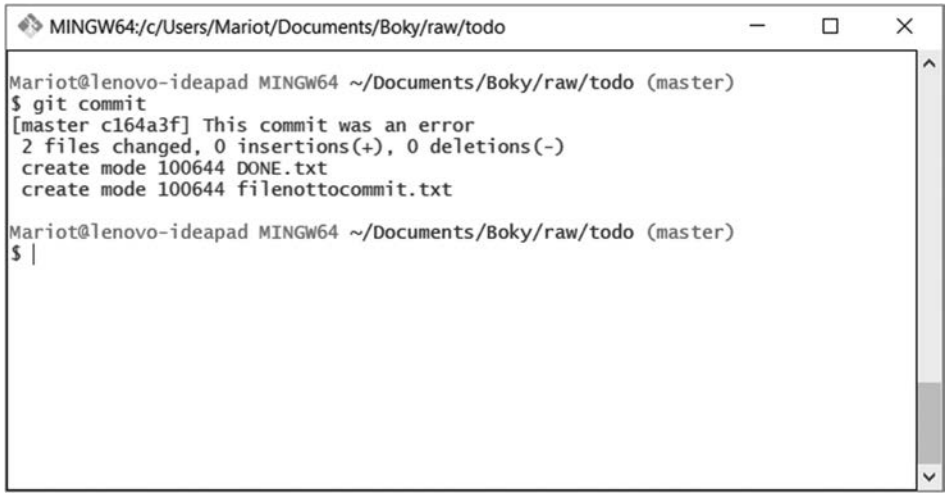


Rysunek 5.8. *Zmodyfikowane i nieśledzone pliki zostały wyróżnione*

Rysunek 5.9. *Komunikat commita jest ostatnim zabezpieczeniem*

Przede wszystkim należy usunąć ostatni commit z projektu poleceniem `git reset`. Użyjemy opcji `--soft`, aby wprowadzone wcześniej zmiany pozostały obecne w katalogu roboczym. `HEAD~1` informuje, że poprzedni commit wykonany na *HEAD* wskazuje aktualny.

```
$ git reset --soft HEAD~1
```



Rysunek 5.10. *Opis commita. Coś sknociliśmy*

Możesz następnie wycofać plik z przechowalni, używając ponownie `git reset`:

```
$ git reset HEAD filenottocommit.txt
```

Sprawdź, czy polecenia zadziałały zgodnie z zamysłem. W tym celu rzuć okiem na aktualny status projektu.

```
$ git status
```

Otrzymasz wynik taki jak na rysunku 5.11.

Jak widzisz, *filenottocommit.txt* nie jest teraz śledzony, ponieważ usunęliśmy go z przechowalni. Oczywiście *fileforgotten.txt* też nie jest śledzony, gdyż nie wysłaliśmy go do przechowalni. Znajduje się w niej jedynie *DONE.txt*, bo nie ruszaliśmy tego pliku po zatwierdzeniu.

■ **Ostrzeżenie** Używaj polecenia `reset` ostrożnie. Jest bardzo niebezpieczne. Koniecznie dokładnie sprawdzaj, co napisałeś.

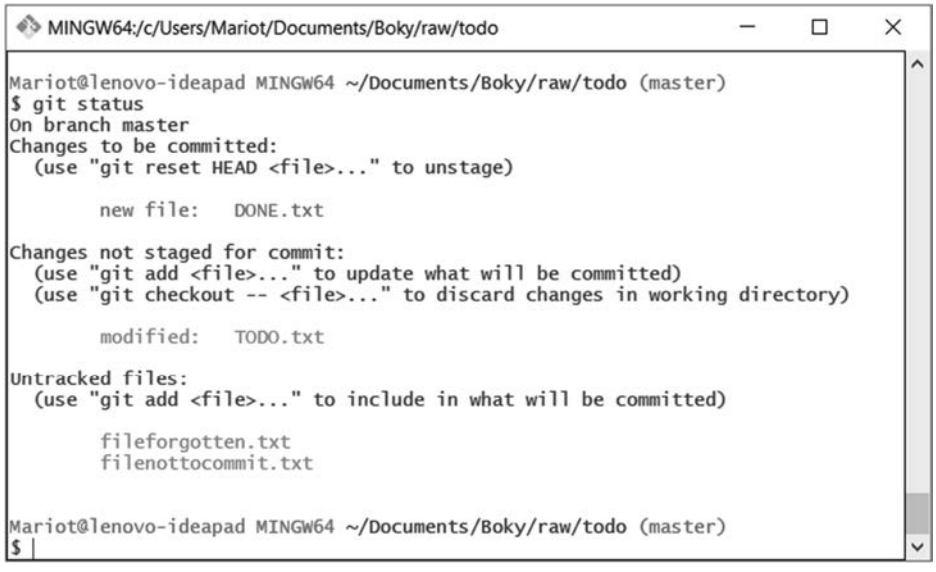
Następnie wyślij do przechowalni właściwy plik:

```
$ git add fileforgotten.txt
```

Skoro już wysłałeś właściwe pliki, możesz zatwierdzić projekt:

```
$ git commit
```

Zamieść w opisie commita jakiś błąd. Przyda się to przy zapoznawaniu się z kolejną funkcją Gita.



Rysunek 5.11. *Status projektu po resecie*

Poprawianie commita

W przypadku zwykłych błędów takich jak pomyłka w opisie commita nie ma potrzeby modyfikowania całego commita. Wystarczy go poprawić. Wypróbujmy to na naszym projekcie!

```
$ git commit --amend
```

Poprawianie wygląda tak samo jak zatwierdzanie zwykłego commita, z tym że opis commita już jest gotowy (rysunek 5.12).

Możesz dowolnie zmienić opis, a następnie standardowo zapisać plik i wyjść z edytora.

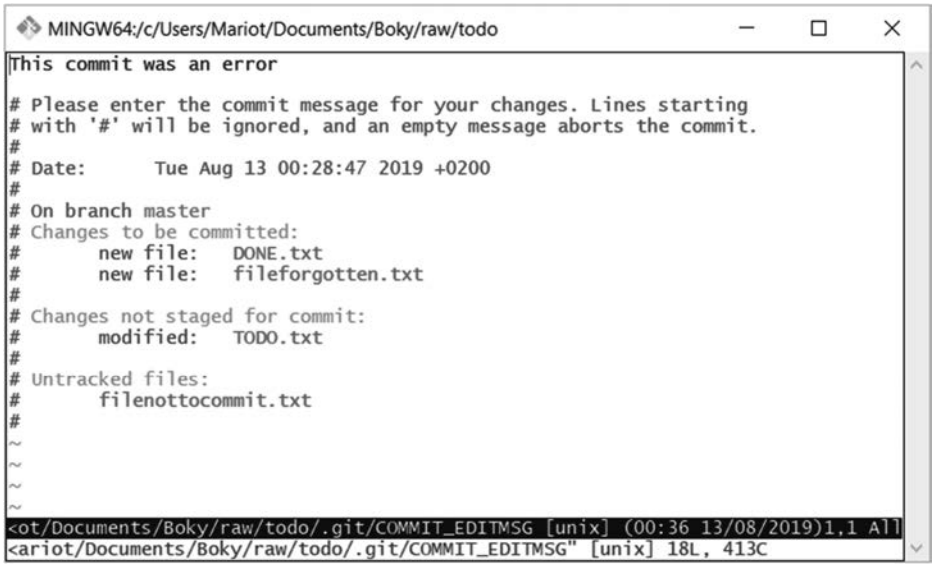
To takie proste! Spójrz na nazwę nowego commita i porównaj ją z nazwą starego. Są różne. Jest tak, ponieważ nazwa commita powstaje w wyniku haszowania informacji zawartych w migawce. Różne stany projektu przekładają się zatem na różne nazwy.

Jeszcze jedna, ostatnia uwaga na temat modyfikowania commitów: nie nadużywaj tej możliwości! Owszem, błędów w kodzie należy się wystrzegać i na ogół dążymy do natychmiastowego ich usuwania. Błędy też jednak pomagają nam się doskonalić; śledzenie własnych błędów sprzyja nauce.

Ćwiczenie: staranne poprawianie commita

Wróć do listy zadań do wykonania. Celem tego ćwiczenia jest staranne poprawienie commita.

- Edytuj kilka plików i wyślij je do przechowalni.
- Zatwierdź je i popełnij błąd w opisie commita.
- Wycofaj plik z przechowalni.
- Wyślij inny plik do przechowalni.
- Popraw commit i wprowadź właściwy opis.



Rysunek 5.12. *Edycja opisu commita*

Podsumowanie

Ten rozdział był poświęcony głównie poruszaniu się po różnych wersjach projektu oraz ich wycofywaniu i poprawianiu. Nie powinieneś mieć już problemów z wprowadzaniem drobnych poprawek do commitów. Koniecznie przeczytaj jeszcze raz pierwszy podrozdział, ponieważ znajdują się w nim informacje kluczowe dla wszystkiego, co robisz w Gicie. Gdy ktoś Cię obudzi w środku nocy, musisz być w stanie wyjaśnić różnice między trzema stanami Gita.

Kolejny rozdział jest krótki, ponieważ poruszymy w nim wyłącznie teorię. Dowiesz się, jak napisać schludny opis commita, co uwzględniać, a co ignorować w commitach, a także poznasz błędy, które najczęściej popełniają początkujący. Koniecznie przeczytaj następny rozdział uważnie, ponieważ zawarta w nim wiedza istotnie pomoże i Tobie, i Twojemu zespołowi. Naprzód!

GitHub. Odzyskaj kontrolę nad wszystkimi wersjami Twoich projektów!

Rozbudowane projekty wymagają zaangażowania zespołu programistów. By ich współpraca była efektywna, niezbędny jest system kontroli wersji, taki jak Git. Umożliwia on tworzenie historii projektu, zarządzanie jego wersjami, przeglądanie wszystkich zmian, a także przywracanie pliku do dowolnej wcześniejszej wersji. Repozytoria Gita można przechowywać w GitHubie — w ten sposób bez ponoszenia dodatkowych opłat udostępnia się kod innym osobom. Korzystanie z Gita i GitHuba nie jest skomplikowane, wymaga jednak pewnej wiedzy i wprawy.

To książka przeznaczona dla programistów, którzy chcą zacząć pracę z Gitem i GitHubem. W każdym rozdziale zawarto wyłącznie przydatne informacje, a te uzupełniono licznymi ćwiczeniami. Dzięki temu równocześnie możesz się uczyć Gita i nabierać sprawności w posługiwaniu się tym systemem. Przewodnik podzielono na trzy części tematyczne dotyczące kontroli wersji, zarządzania projektami i pracy zespołowej. To ułatwi Ci wdrożenie się do rzeczywistej pracy i rozwiązywanie problemów. Poznasz zasady planowania i realizacji projektów z GitHubem, a także wypróbujesz sposoby rozstrzygania konfliktów scalania, co sprawi, że poczujesz się pewniej w pracy zespołowej w profesjonalnym środowisku.

W książce:

- czym są, do czego służą i jak działają systemy kontroli wersji
- jak przygotować Git do pracy i jak ją rozpocząć
- obieg pracy w GitHubie: zgłoszenia, gałęzie, pull requesty
- konflikty scalania, ich rozstrzyganie i zarządzanie zmianami kodu
- najprzydatniejsze narzędzia GUI Gita
- zaawansowane polecenia Gita i rozwiązywanie częstych problemów

Mariot Tsitoara programuje w Pythonie i JavaScripcie. Od 2015 roku jest przedstawicielem Mozilli. Często uczestniczy w konferencjach technicznych, na których wypowiada się na temat otwartego kodu i nowych technologii, takich jak Rust, WebVR i WebAssembly. Mieszka w Bordeaux, zajmuje się tworzeniem kodu niewielkich specjalistycznych narzędzi edukacyjnych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
INFORMATYKA W NAJLEPSZYM WYDANIU

ISBN 978-83-283-8735-5
9 788328 387355
Cena: 69,00 zł
Apress